1 Project Description

In this project we were asked to complete the following tasks: acquire the required files (which includes some new test images and videos as well as source files that incorporate a calculation of the Peak Signal-to-Noise Ratio (PSNR)), test and modify the latest scripts (which requires that we change the location output images may be saved, optionally write a script to automate the input of different files run on different sets of parameters, and test the code to see if we can acquire the average PSNR from different input sources), understand the PSNR calculation (which asked that we read through some reference material related to the topic of the calculation), optimize the edge detection program using a variety of parameters and multi-processing/threading methods, and finally save the output images or videos with the best and worst PSNR and frames-per-second(FPS) calculations.

With this in mind I expect that I will understand the particulars of the PSNR calculation equations, I will be able to confidently discuss the reason why certain outputs obtained through the use of certain relevant input values performed better or worse with respect to PSNR or FPS, and that I will be able to devise a way in which to automate the process of testing the variety of input images/videos. I, of course, expect that multi-threading the programs will improve the FPS I will achieve, but due to my lack of previous experience with PSNR I am uncertain of the results. PSNR is a comoparitor quantity and the output of the edge detection algorithm is very dissimilar to kind of colored input image.

2 Experimental Setup

2.1 Methodology

I once again relied on my previous experience to complete this project and it did not let me down. I knew that automating the program with different input images and input parameters could be done in a few different ways: all in the cpp file, all in a bash script, or some hybrid of the two. I opted for a hybrid of the two, because it kept each file cleaner and easier to read. For research on topics like PSNR I referred heavily to the supplied literature. I made great use again of the GitHub repository that I had previously set up to transfer my coding changes to the relevant directories on the Pi, but this time also made the communication full-duplex in that I had to send output files back from the Pi to my host machine so that I could use my experience with Microsoft Excel to analyze the outputs and find the extrema. For any problems with the Pi I was sure I could ask my classmate Richard, who lent me the Pi, and for any problems with Excel I would ask my classmate Elina.

2.2 Process/Procedure

(1) I first downloaded the given code and test images/videos. I specifically noted the inclusion of the *calcpsnr.c* file and looked inside of it to verify that the equation mentioned in the project specifications. After verifying that the MSE code was correct I looked at the psnr calculation and you can see the similarity between what was given in the spec and what is calculated.

```
double psnr(double mse)
{
  return 10 * log10(255 * 255/mse);
}
```

$$\begin{aligned} PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\ &= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\ &= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE) \end{aligned}$$

(2) Next I moved on to just running the code on the input tiger image, the ground control video, and captures from the picam. All things went smoothly and I received teh edge detection outputs along with the relevant PSNR values. More on this later.

(3) After, I began making changes to the *camera_canny_psnr.cpp* file to allow for user specified input parameters. Some of the changes I made can be seen below:

```
// #define PICAM_WIDTH 640
// #define PICAM_HEIGHT 480

// #define GROUND_WIDTH 1280
// #define GROUND_HEIGHT 720

// #define TIGER_WIDTH 888
// #define TIGER_HEIGHT 900

//#define NFRAME 30.0

enum IMGSRC {PICAM, GROUND, TIGER};
```

Figure 1: remove macros and define enum

```
enum IMGSRC img_src = PICAM;
int mp;
float NFRAME = 30.0;
```

Figure 2: initialize the enum

```
if(argc < 8){
  fprintf(stderr,"\n<USAGE> %s sigma tlow thigh imgsrc [writedirim]\n",argv[0]);
  fprintf(stderr,"      sigma:      Standard deviation of the gaussian");
  fprintf(stderr," blur kernel.\n");
  fprintf(stderr,"      tlow:       Fraction (0.0-1.0) of the high ");
  fprintf(stderr,"edge strength threshold.\n");
  fprintf(stderr,"      thigh:      Fraction (0.0-1.0) of the distribution");
  fprintf(stderr," of non-zero edge\n                  strengths for ");
  fprintf(stderr,"hysteresis. The fraction is used to compute\n");
  fprintf(stderr,"                  the high edge strength threshold.\n");
  fprintf(stderr,"      imgwidth:    integer 400 850 1300\n");
  fprintf(stderr,"      imgheight:   integer 100 550 1000\n");
  fprintf(stderr,"      multi-thread:integer 0=none, 1=pt, 2=omp\n");
  fprintf(stderr,"      imgsrc:      integer 0=picam, 1=ground_ctrl, 2=tiger\n");
  fprintf(stderr,"      writedirim: Optional argument to output ");
  fprintf(stderr,"a floating point");
  fprintf(stderr," direction image.\n\n");
  exit(1);
```

Figure 3: additions to useage error

```
sigma = atof(argv[1]);
tlow  = atof(argv[2]);
thigh = atof(argv[3]);
cols  = atoi(argv[4]);
rows  = atoi(argv[5]);
mp    = atoi(argv[6]);
```

Figure 4: grab values from argv

```
img_src = (enum IMGSRC)atoi(argv[7]);
switch (img_src) {
case PICAM:
  if ( NULL == (fout = fopen("picam.csv", "a"))) {
    printf("Failed to open picam.csv\n");
    return 0;
  }
  if(!cap.open(0)){
    printf("Failed to open media\n");
    return 0;
  }
  break;
case GROUND:
  NFRAME = 30.0;
  if ( NULL == (fout = fopen("ground.csv", "a"))) {
    printf("Failed to open ground.csv\n");
    return 0;
  }
  if(!cap.open("ground_crew.h264")){
    printf("Failed to open media\n");
    return 0;
  }
  break;
case TIGER:
  NFRAME = 1.0;
  if ( NULL == (fout = fopen("tiger.csv", "a"))) {
    printf("Failed to open tiger.csv\n");
    return 0;
  }
  if(!cap.open("tiger_face.jpg")){
    printf("Failed to open media\n");
    return 0;
  }
  break;
default:
  if ( NULL == (fout = fopen("picam.csv", "a"))) {
    printf("Failed to open picam.csv\n");
    return 0;
  }
  if(!cap.open("picam.h264")){
    printf("Failed to open media\n");
    return 0;
  }
  break;
}
```

Figure 5: the switch statement that handled different image sources

```
sprintf(outfilename, "edge/EDGE_%03d.pgm", count);
```

Figure 6: changed edge output dir

```
sprintf(outfilename, "raw/RAW_%03d.pgm", count);
```

Figure 7: changed raw output dir

```
snprintf(line, len, "%3.2f, %3.2f, %3.2f, %i, %i, %i, %lf, %4f, %3.2f\n",
         sigma, tlow, thigh, cols, rows, mp,
         time_elapsed/1000000, NFRAME/(time_elapsed/1000000), PSNR);
if ( fputs(line, fout) < 0 ) {
  printf("Error appending line to csv file\n");
  return 0;
}
```

Figure 8: show write to csv command

It is quite important to note how I decided to store the outputs of the programs. First I set up three csv files to store the outputs and devised a column title row to organize the data. Then I used the *snprintf* function to toss all of the necessary information, separated by commas, into the char array which would then be written to the relevant csv file.

(4) In addition to the changes I made to the *camera_canny_psnr.cpp* file I also created a bash script to call which ran the respective executable outputs of the following compilation commands (note that I had to copy the pthread and omp code from my previous projects in order to run these versions of the edge detection programs):

```
g++ canny_util.c calcpsnr.c camera_canny_psnr.cpp -o psnr -I. /
`pkg-config --cflags --libs opencv4`

g++ canny_util_pt.c calcpsnr.c camera_canny_psnr.cpp -o psnr_pt -I. /
`pkg-config --cflags --libs opencv4` -lpthread

g++ canny_util_omp.c calcpsnr.c camera_canny_psnr.cpp -o psnr_omp -I. /
`pkg-config --cflags --libs opencv4` -fopenmp -lm
```

```
#!/bin/bash

#Remember! This script takes the
#img id 0, 1 ,2, 3 as input=$1

# 3^6 different outputs
sigma=(0.6 1.5 2.4)
tlow=(0.2 0.35 0.5)
thigh=(0.6 0.75 0.9)
width=(400 850 1300)
height=(100 550 1000)
#MP    : 1x, pthread, openmp
c=0
```

Figure 9: initialize different input parameters

I think it's important to highlight what I did in this script to allow me to easily toggle input parameters as well as verify that the program had finished all of the necessary iterations.

- Observe that I used bash arrays to store the possible input parameter values and named them accordingly. This allowed me to add and remove relevant parameters so as to better ascertain the effect that each might have on the final FPS and PSNR values.
- Note that I did not use an array to store the different executable options. I felt that doing thing would dramatically impair the readability of this script and, as you can see, each iteration, given some input

```
for sig in ${sigma[@]};
do
    for low in ${tlow[@]};
    do
        for high in ${thigh[@]};
        do
            for w in ${width[@]};
            do
                for h in ${height[@]};
                do
                    #assume that they
                    #have already been compiled
                    ./psnr $sig $low $high $w $h 0 $1
                    ./psnr_pt $sig $low $high $w $h 1 $1
                    ./psnr_omp $sig $low $high $w $h 2 $1
                    ((c++))
                    echo "Finished iteration $c"
```

Figure 10: loop through different input parameters

parameters, will run all three possible executables and store their output in a csv file as mentioned previously.

- Arguably the most important observation one could make regarding this script is possibly the least impressive thing, that of the bash variable *c*. *c* provided me with a way to keep track of the script's progress as it ran through the many different parameter options. These programs took a non-negligible amount of time to run and I was able to test the Pi's ability to run overnight without me watching it and I was able to verify that the programs had successfully completed with output like below.

```
Finished iteration 237
Finished iteration 238
Finished iteration 239
Finished iteration 240
Finished iteration 241
Finished iteration 242
Finished iteration 243
```

Figure 11: Finished iteration counter

(5) Next, I scanned through the supplementary materials provided regarding the PSNR calculation.

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(I(i,j) - K(i,j))^2$$

$$PSNR = 10\log_{10}\left(\frac{MAX_I^2}{MSE}\right)$$

The PSNR is a measure of ratio of the maximum value of an input signal, I, and the residual error between the ideal signal, I, and the noisy signal, K. Measured in decibels, it gives us an idea about how similar two signals are and for that reason is a great quantification of the success of a decompression.

(6) Subsequently, I uploaded the generated csv files to my GitHub repository with the commands:

```
git add tiger.csv ground.csv picam.csv

git commit

git push origin main
```

Note that to generate the picam csv file I first captured a bunch of images, with a single run of the program. Then I stitched the raw images together into a video with

```
ffmpeg -i raw/RAW_%03d.pgm -pix_fmt yuvj420p picam.h264
```

and used that video as input to the picam image source parameter. In this way I would speed up the running time of the program (because I would not be capturing new images with each run), but I would also be receiving more reliable results because my outputs to the csv file would be based on a single video and not many.

(7) With the csv files generated I pulled them down from the repo with a *git pull origin main* and opened them with Microsoft Excel on my host machine.

(8) Within Microsoft Excel I was able to generate the minimums and maximums of the columns of FPS and PSNR data that I had generated and then compare those values to the same ones generated from different input files. More on this later.

3 Results

The results of this project will be in the form of pictures depicting the analyses of the csv file outputs. As mentioned previously, the csv files were treasure troves on informative data which I was able to use to find the extrema related to FPS and PSNR for each input image/video type (tiger, ground control, and picam). I will also, if I have time, generate the respective extreme PSNR/FPS images and videos and include them in the zip file submission for this project.

Below we note the values of the min and max PSNR and FPS for each of the input image sources (tiger, ground control, and picam). These are obtained through the use of the *MIN()* and *MAX()* functionalities offered by Microsoft Excel which allowed me to easily scan the roughly 750 different columns worth of data. In order to get the corresponding input parameters for the respective input images I would highlight the column containing the extrema value I wanted to find, and then used Excel's find funcationality to isolate the row which contained the value. I include these parameters in a table below.

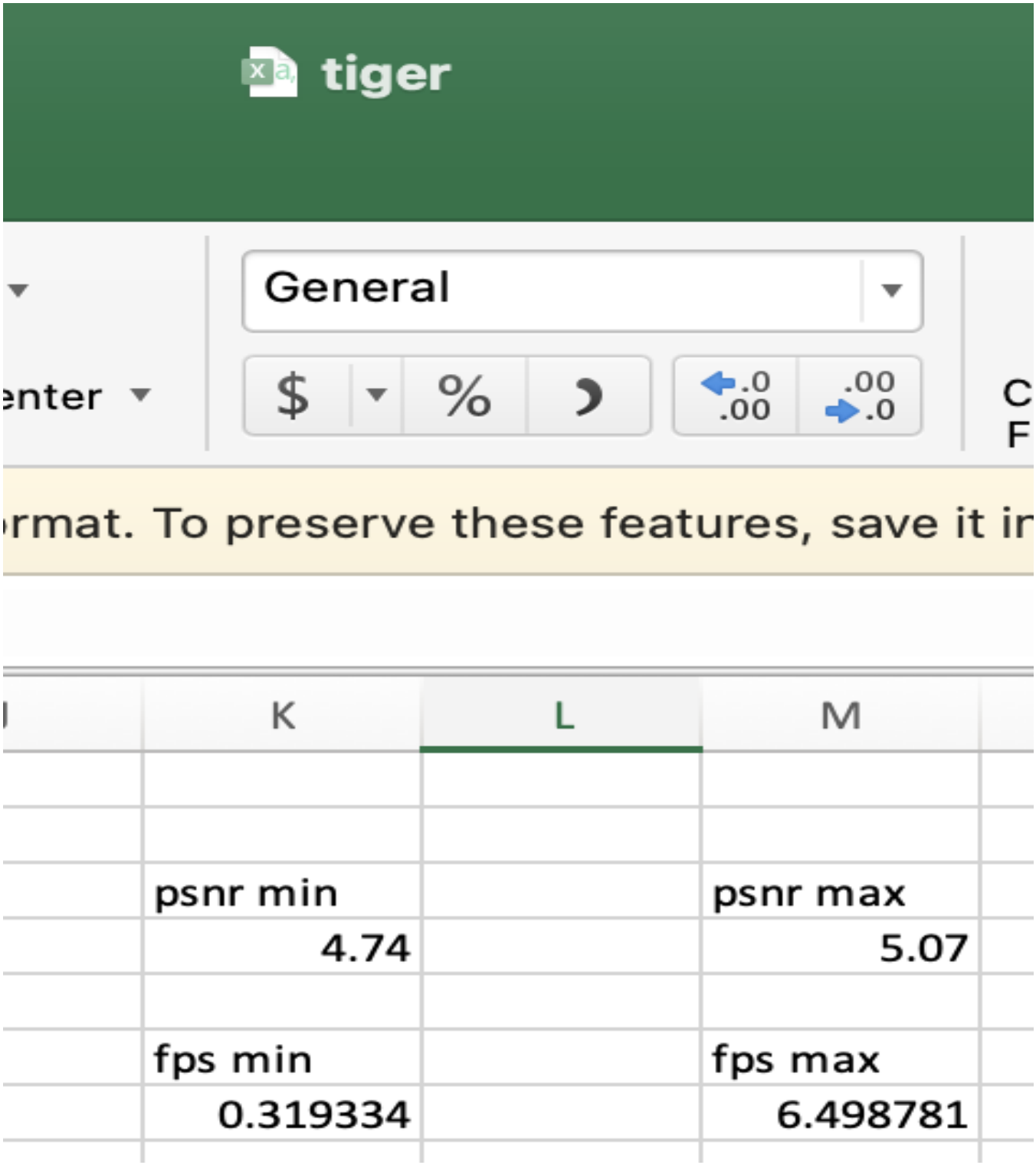

Figure 12: Tiger PSNR/FPS Extrema

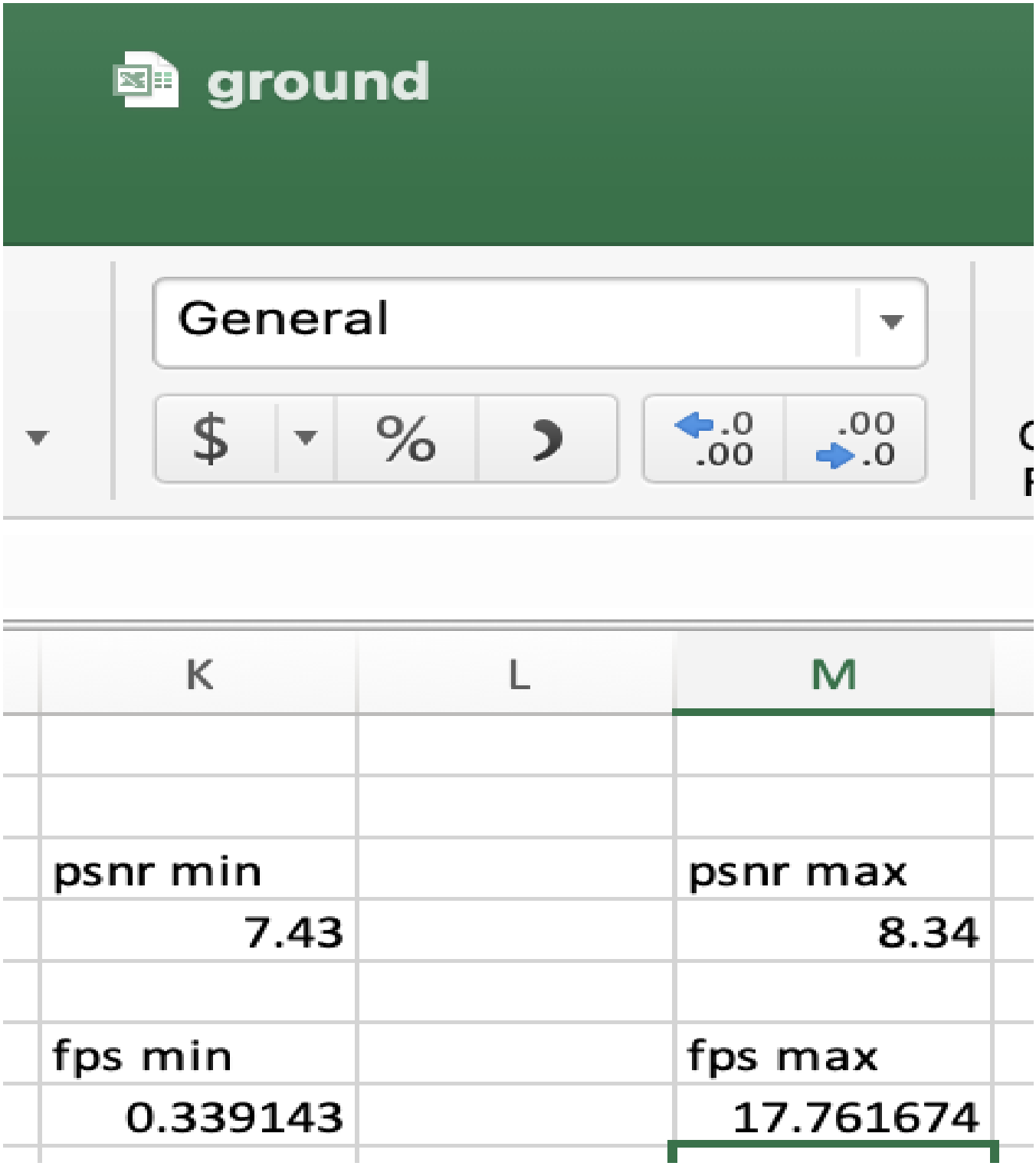

Figure 13: Ground control PSNR/FPS Extrema

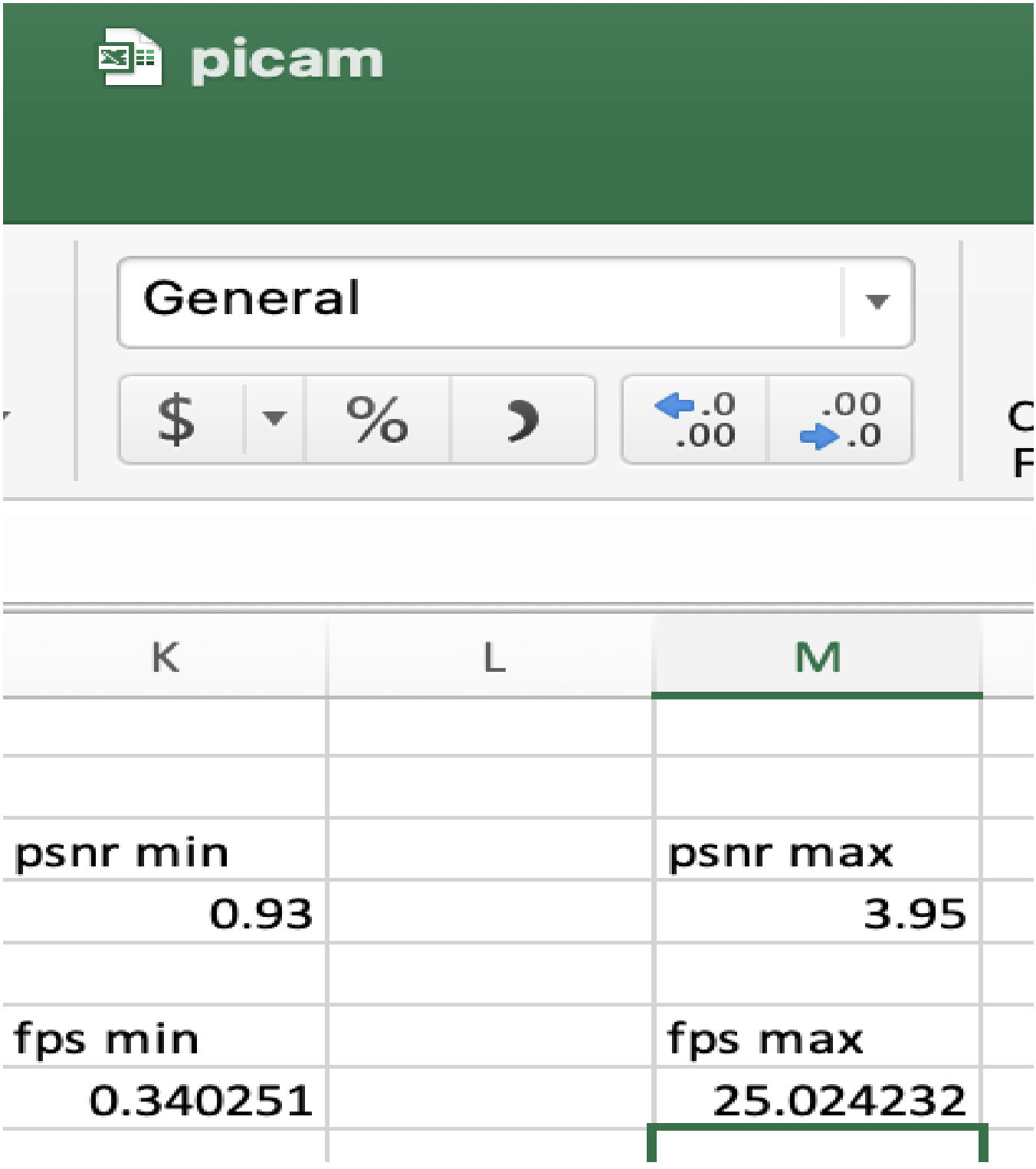

Figure 14: Picam PSNR/FPS Extrema

sigma	tlow	thigh	width	height	multi-thread option	time	fps	psnr
2.4	0.5	0.9	1300	100	2	0.316413	3.160426	4.74
0.6	0.35	0.6	400	100	2	0.208026	4.807091	5.07
2.4	0.5	0.6	1300	1000	0	3.131513	0.319334	4.83
0.6	0.5	0.9	400	100	1	0.153875	6.498781	4.88

Tiger min/max PSNR/FPS Respectively

sigma	tlow	thigh	width	height	multi-thread option	time	fps	psnr
0.6	0.2	0.6	1300	1000	2	29.988782	1.000374	7.43
0.6	0.5	0.9	400	100	2	1.697707	17.670894	8.34
2.4	0.5	0.6	1300	1000	0	88.458155	0.339143	8.13
0.6	0.5	0.9	400	100	1	1.68903	17.761674	8.34

Ground Control min/max PSNR/FPS Respectively

sigma	tlow	thigh	width	height	multi-thread option	time	fps	psnr
1	0.2	0.6	640	480	0	16.385986	1.830833	0.93
0.6	0.2	0.6	400	100	0	1.795192	16.711304	3.95
2.4	0.5	0.6	1300	1000	0	88.170194	0.340251	3.67
0.6	0.35	0.9	400	100	1	1.198838	25.024232	3.73

Picam min/max PSNR/FPS Respectively

The following are graphs of the various canny parameters versus the calculated PSNR.

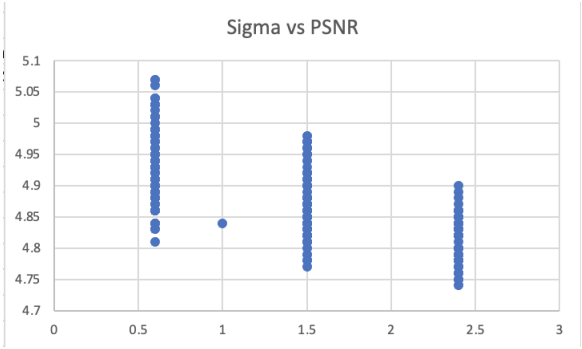

Figure 15: tiger sigma vs psnr

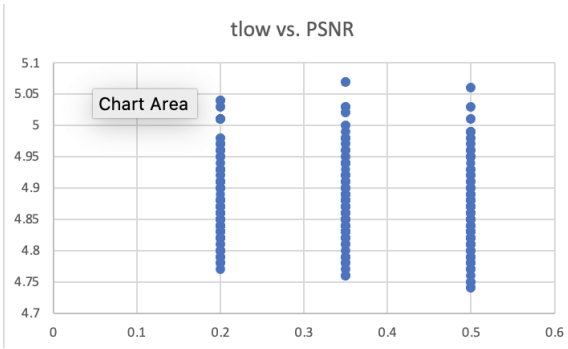

Figure 16: tiger tlow vs psnr

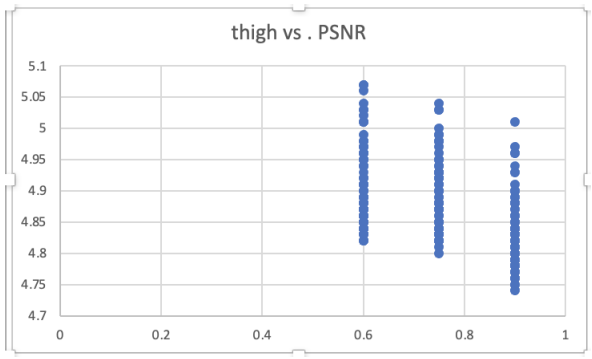

Figure 17: tiger thigh vs psnr

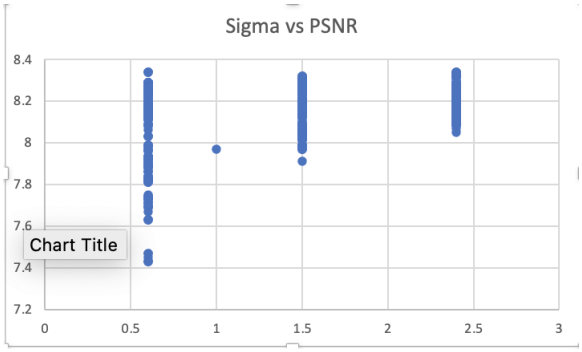

Figure 18: gound sigma vs psnr

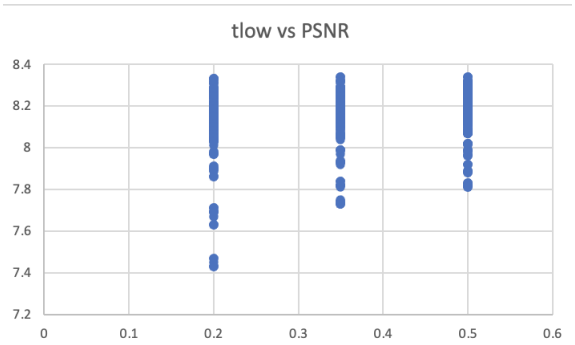

Figure 19: gound tlow vs psnr

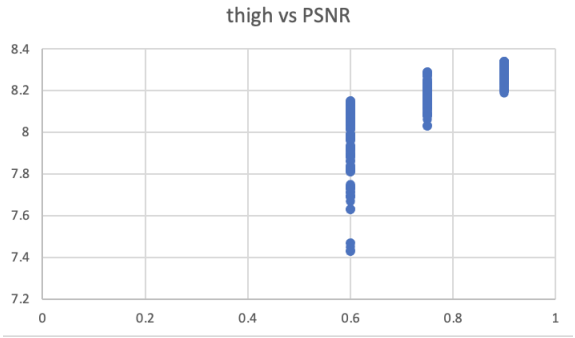

Figure 20: gound thigh vs psnr

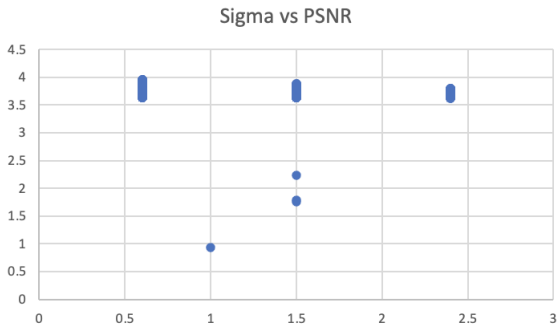

Figure 21: picam sigma vs psnr

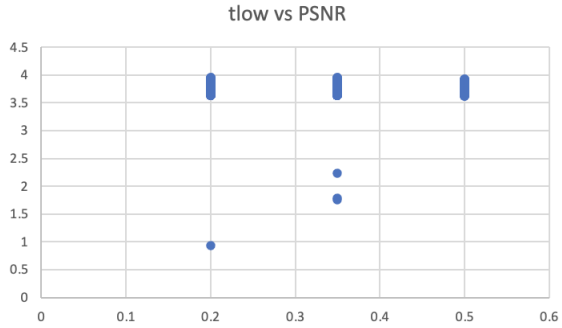

Figure 22: picam tlow vs psnr

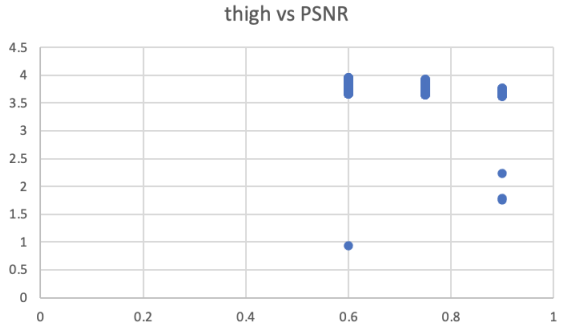

Figure 23: picam thigh vs psnr

Discussion of these results will occur later.

4 Problems and Discussion

4.1 Problems

Problem
I didn't really have any problems. At first I was worried that I wouldn't be able to run the programs on my Pi fast enough to get enough data to provide meaningful conclusions.

Solution But once I was able to confirm that the Pi could continue running programs on mine overnight I was able to generate pretty much all the data I needed!

4.2 Discussion

4.2.1 Differences in performance

As we can see from the tables and graphs, we received some pretty interesting results:

- Regarding the tiger image and picam video, we notice that the lowest value of thigh gave us the highest value of PSNR. While in contrast the highest value of thigh gave us the highest value of PSNR for the ground control video. I would say that this primarily has to do with the variance in each frame when comparing the tiger and picam to the ground control video. The tiger image is more uniform and stays the same throughout and the picam video, when I took it remotely, was of a still bedroom with no moving objects. The ground control video by contrast had a significant deal of movement and by raising the value of thigh we were able to limit the edges found and match with still objects like the background and sky better. The tiger image and picam video didn't change much and thus a lower thigh value with more edges did not effect the PSNR value nearly as much.
- We also note the obvious conclusion that larger images yielded the lowest FPS as well as the non-multi-threaded options, while the smaller multi-threaded options gave us the highest FPS each time.
- We also note that, across the board, larger images gave us lower PSNR values while smaller ones gave us higher values. I would conclude that the reason for this is more of a sample size issue. There are fewer comparisons to make between smaller images and the larger ones and thus the magnitude of the differences, though relatively similar is absolutely very different.
- I also note that high FPS seemed to correlate with higher PSNR. Higher FPS should be able to capture more detail and clearly this detail has a positive effect on the PSNR value.
- We also observe that lower sigma values correlated with higher PSNR. A wider kernel should make fewer edges and thus smooth out the output. This is probably the reason for this correlation.

5 Conclusion

This project was an interesting exploration of different parameters. This is really the first project where we had the opportunity to really compare outputs with a varying number of parameters and it forced us to push the limits of our Raspberry Pi to generate them. Generally, PSNR correlates with sigma directly, while thigh does so differently depending on the input. PSNR correlates directly with image size and FPS does so as well. FPS also correlates with image size directly.